CS305 – Programming Languages
Fall 2024-2025

HOMEWORK 4

Implementing a Semantic Analyzer for LCD using Scheme

Due date: December 24, 2024 @ 23:55

NOTE

Only SUCourse submission is allowed. No submission by e-mail. Please see the note at the end of this document for late submission policy.

1 Introduction

In this homework, you will implement a tool that includes a simple semantic analyzer for LCD language using Scheme. For more detailed information about the homework read the rest of the document.

The tool that you will implement in this homework will have various functions. In these functions, the tool will perform some semantic checks.

2 LCD Language format in Scheme

You are familiar with the LCD language from the previous homeworks. The Scheme functions that you will implement will take an LCD program as a parameter and perform some analysis as explained in this document.

Similar to our exercises in the class, LCD program will be represented as a list. We slightly changed the syntax of LCD language for this purpose. We defined the LCD Language format in Scheme as a nested list where the first element of the nested list includes Declarations, the second element includes Assignments and the last element includes Evaluation. As in previous assignments, declaration, assignment and evaluation are used in that order, and the use of parentheses in the Assignment section is essential. The example program that is defined in the Scheme can be found below.

NOTE: You can assume that the LCD program will be grammatically correct. No need for you to check the grammar.

```
(define example-program
   '(
    ;; Declarations
    (("input" input1 input2)
     ("node" node1 node2)
     ("output" output1 output2)
     ("input" input3 input4))

    ;; Assignments
    ((node1 = input3 or (not input4))
     (node2 = node1 and input2)
     (output1 = input1 and node2)
     (output2 = node4 or node2)
     (output3 = (node8 xor input2) and (not input3)))

    ;; Evaluation
    (("evaluate" circuit1 ((input1 true) (input2 true)))
     ("evaluate" circuit2 ((input2 false) (input3 false)))
     ("evaluate" circuit3 ((input2 true) (input1 false)))
     ("evaluate" circuit4 ((input2 false) (input4 true))))
  ))
```

3 Semantic Rules

Your semantic analyzer should start by performing an analysis for the following semantic rules. For each violation of these rules, your semantic analyzer must print the identifiers that violate the rules.

Rule 1 (Undeclared Identifiers): *Every identifier that is used in either the circuit design block or the evaluations block must be declared in the declarations block.*

In the circuit design block or the evaluations block, we have identifiers that are used. All such identifiers must be declared in the declarations block. Note that this includes the identifiers used on the left-hand side, and also the identifiers used in the expressions on the right-hand side of the assignments.

Rule 2 (Multiple Declarations): *An identifier can be declared only once.*

In the declarations block, identifiers to be used in the program are introduced. An identifier can be introduced only once, and it has to be as an input, or as an output, or as a node. Therefore, having an identifier being declared multiple times is not allowed even when these declarations are of different types (input, output, node). For example, if an identifier is declared as an input and also as a node, this is still an error.

Rule 3 (Unused Inputs): *Every input must be used.*

All input identifiers introduced in the declarations block must be used somewhere

in the circuit design block. In other words, if an identifier is declared as an input, then this identifier has to appear within the expression on the right-hand side of some assignment given in the circuit design block.

Rule 4 (Unassigned Nodes and Outputs): *Every identifier that is defined in the declarations block as a node or as an output must be assigned.*

If an identifier is declared as a "node" or as an "output" in the declarations block, it has to be assigned in the circuit design block. In other words, it has to appear on the left-hand side of an assignment in the circuit design block.

Rule 5 (Multiple Assignment to Nodes and Outputs): *An identifier that is defined in the declarations block as a node or as an output cannot be assigned multiple times.*

For an identifier that is declared as a "node" or as an "output" in the declarations block, there cannot be multiple assignment statements that assign to this identifier. Therefore, we can have only one assignment statement where we see this identifier on the left-hand side.

Rule 6 (Unassigned Inputs): *Every input identifier has to be assigned in each circuit evaluation statement.*

If an identifier is declared as an "input" in the declarations block, then in each circuit evaluation this identifier must be assigned to a value (which will be a constant boolean value, as guaranteed by the grammar).

Rule 7 (Multiple Assignment to Inputs): *An input cannot be assigned multiple times in an evaluation.*

If an identifier is declared as an "input" in the declarations block, then in a circuit evaluation statement, there cannot be multiple assignments to this identifier.

Rule 8 (Incorrect Assignments to Inputs): *An input can only be assigned in the circuit evaluations.*

If an identifier is declared as an "input" in the declarations block, then it cannot be assigned in the circuit design block.

Rule 9 (Incorrect Assignments to Nodes and Outputs): *A node or an output can only be assigned in the circuit design block.*

If an identifier is declared as a "node" or as an "output" in the declarations block, then it cannot be assigned in a circuit evaluation statement.

4 Functions

You need to define various functions that include the combination of the rules described above. Define each function explained below separately.

find-undeclared-identifiers: *Implement Rule 1 and return undeclared identifiers in the same order defined in the program.*

Example:

```
(define example-program
    '(
    ;; Declarations
    (("input" input1 input2)
     ("node" node1 node2)
     ("output" output1 output2))

    ;; Assignments
    ((node1 = input3 or (not input4))
     (node2 = node1 and input2)
     (output1 = input1 and node2)
     (output2 = node4 or node2)
     (output3 = (node8 xor input2) and (not input3)))

    ;; Evaluation
    (("evaluate" circuit1 ((input1 true) (input2 true)))
     ("evaluate" circuit2 ((input2 false) (input3 false)))
     ("evaluate" circuit3 ((input2 true) (input1 false)))
     ("evaluate" circuit4 ((input2 false) (input4 true))))
  ))

  (find-undeclared-identifiers example-program)
```

The output should be the following:

```
(input3 input4 node4 output3 node8 input3 input3 input4)
```

If there are no errors of this kind, then the function would just return an empty list.

find-multiple-declarations: *Implement Rule 2 and return multiple declared identifiers in the same order defined in the program.*

Example:

```
(define example-program
  '(
    ;; Declarations
    (("input" input1 input2)
     ("node" node1 node2)
     ("input" input1 input2)
     ("output" output1 output2)
     ("node" output1 output2)
     ("node" node2 node1 input1))

    ;; Assignments
    ((node1 = input3 or (not input4))
     (node2 = node1 and input2)
     (output1 = input1 and node2)
     (output2 = node4 or node2)
     (output3 = (node8 xor input2) and (not input3)))

    ;; Evaluation
    (("evaluate" circuit1 ((input1 true) (input2 true)))
     ("evaluate" circuit2 ((input2 false) (input3 false)))
     ("evaluate" circuit3 ((input2 true) (input1 false)))
     ("evaluate" circuit4 ((input2 false) (input4 true))))
  ))

  (find-multiple-declarations example-program)
```

The output should be the following:

```
(input1 input2 output1 output2 node2 node1 input1)
```

If there are no errors of this kind, then the function would just return an empty list.

check-identifier-usage: *Implement Rule 3, Rule 4 and Rule 5 and return a nested list. The first element is, a list of the Unused Inputs that are same order as a program. The second element is a list of Unassigned Nodes and Outputs that are in the same order as the program. The third and the last element is a list of Multiple Assignment to Nodes and Outputs that are the same order as a program where first repeated will be in the beginning.*

Example:

```
(define example-program
  '(
    ;; Declarations
    (("input" input1 input2 input4)
     ("output" output1 output2 output3)
     ("node" node1 node2 node4 node3)
     ("input" input3))

    ;; Assignments
    ((node1 = input1 or (not input2))
     (node2 = node1 and input2)
     (output1 = input1 and node2)
     (output2 = node1 or node2)
     (output2 = (node1 xor input2) and (not input1))
     (node2 = node1 and input2))

    ;; Evaluation
    (("evaluate" circuit1 ((input1 true) (input2 true)))
     ("evaluate" circuit2 ((input2 false) (input3 false)))
     ("evaluate" circuit3 ((input2 true) (input1 false)))
    )
))

  (check-identifier-usage  example-program)
```

The output should be the following:

```
((input4 input3) (output3 node4 node3) (output2 node2))
```

In case there are no errors of a particular kind, the corresponding sublist should be empty. For example, if there are no unassigned nodes and outputs, the second element of the returned list would be an empty list. If there are no errors of any kind, then it would return all sublists as empty lists, i.e. `( ( ) ( ) ( ) )`

check-inputs-in-evaluation: *Implement Rules 6 and 7 and return a nested list. The first element is a list of evaluation identifiers with unassigned inputs. The second element is a list of evaluation identifiers with multiple assignments to an input. Return the evaluation names in the same order as they appear in the program.*

Example:

```
(define example-program
  '(
    ;; Declarations
    (("input" input1 input2)
     ("output" output1 output2)
     ("node" node1 node2 node4 node3)
     ("input" input3))

    ;; Assignments
    ((node1 = input1 or (not input2))
     (node2 = node1 and input2)
     (output1 = input1 and node2)
     (output2 = node1 or node2))

    ;; Evaluation
    (("evaluate" circuit3 ((input1 true) (input2 true)))
     ("evaluate" circuit2 ((input2 true) (input1 false) (input3 true)))
     ("evaluate" circuit1 ((input2 false) (input2 true) (input3 false)))
    )
))

  (check-inputs-in-evaluation  example-program)
```

The output should be the following:

```
((circuit3 circuit1) (circuit1))
```

In case there are no errors of a particular kind, the corresponding sublist should be empty. For example, if there are no evaluation statements with unassigned inputs, the first element of the returned list would be an empty list. If there are no errors of either kind, then it would return both sublists as empty lists, i.e. `( ( ) ( ) )`

check-incorrect-assignments: *Implement Rules 8 and 9 and return a nested list. The first element is a list of incorrectly assigned inputs in the circuit design block. The second is a list of incorrectly assigned nodes and outputs in evaluations. Return the identifier names in the same order as the program.*

Example:

```
(define example-program
  '(
    ;; Declarations
    (("input" input1 input2 input3)
     ("node" node1 node2)
     ("output" output1 output2))

    ;; Assignments
    ((node1 = input1 or (not input2))
     (node2 = node1 and input2)
     (input2 = node1 or  node2)
     (input1 = node2 and (not node1))
     (input2 = node2 and (not node1))
     (output1 = input1 and node2)
     (output2 = (node1 xor input2) and (not node2)))

    ;; Evaluation
    (("evaluate" circuit1 ((output2 false) (input2 false) (node1 false)))
     ("evaluate" circuit2 ((input2 true) (node2 false) (output1 true)))
    )
))

  (check-incorrect-assignments  example-program)
```

The output should be the following:

```
((input2 input1 input2) (output2 node1 node2 output1))
```

In case there are no errors of a particular kind, the corresponding sublist should be empty. For example, if there are no inputs assigned in the circuit design block, the first element of the returned list would be an empty list. If there are no errors of either kind, then it would return both sublists as empty lists, i.e. `( ( ) ( ) )`

5 How to Submit

Submit your Scheme file named as username-hw4.scm where username is your SU-Course username. In this file you must have scheme procedures defined with the names given in Section 4. In other words, you must have

```
(define find-undeclared-identifiers (lambda (lcd) ... ))
(define find-multiple-declarations  (lambda (lcd) ... ))
(define check-identifier-usage      (lambda (lcd) ... ))
(define check-inputs-in-evaluation  (lambda (lcd) ... ))
(define check-incorrect-assignments (lambda (lcd) ... ))
```

where the parts shown as `...` represent the parts where you will implement your procedures.

You can have additional procedures that you use within the code, but we will only test/grade by using the procedures given above.

6 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points otherwise]
- **Important**: The golden is shared as the file

  `onur.dogan/cs305/hw4-golden.com`

  on `cs305.sabanciuniv.edu`

  You can access the functions to be implemented by loading this file as

  `(load "hw4-golden.com")`

  You can also find some example LCD programs given by the syntax used in this homework in the same directory.
- **Important: Since this homework is evaluated automatically make sure your output is exactly as it is supposed to be. Some of the points that we can think of are:**
  - **Not having an empty list or empty sublists as required when there are no errors**
  - **Not paying attention to the order of the elements of the lists**
- If you are not sure about your outputs you can compare your outputs with the outputs given by the golden.

- You may get help from our TA or from your friends. However, **you must implement the homework by yourself**.
- Start working on the homework immediately.
- If you develop your code or create your test files on your own computer (not on cs305.sabanciuniv.edu), there can be incompatibilities once you transfer them to the cs305 machine. Since the grading will be done automatically on the cs305 machine, we strongly encourage you to do your development on the cs305 machine, or at least test your code on the cs305 machine before submitting it. If you prefer not to test your implementation on the cs305 machine, this means you accept to take the risks of incompatibilities. Even if you may have spent hours on the homework, you can easily get 0 due to such incompatibilities.

LATE SUBMISSION POLICY

Late submission is allowed subject to the following conditions:

- Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
- If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
- If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
- We will not accept any homework later than 500 mins after the deadline.
- SUCourse's timestamp will be used for STF computation.
- If you submit multiple times, the last submission time will be used.